Bit-Level Transformation and Optimization for Hardware Synthesis of Algorithmic Descriptions

Jiyu Zhang*[†], Zhiru Zhang[+], Sheng Zhou[+], Mingxing Tan*, Xianhua Liu*, Xu Cheng*, Jason Cong[†]

*MicroProcessor Research and Development Center, Peking University, Beijing, PRC

[†] Computer Science Department, University Of California, Los Angeles, CA 90095, USA

[+]AutoESL Design Technologies, Los Angeles, CA 90064, USA

{zhangjiyu, tanmingxing, liuxianhua, chengxu}@mprc.pku.edu.cn

{zhiruz, zhousheng}@autoesl.com, cong@cs.ucla.edu

ABSTRACT

As the complexity of integrated circuit systems increases, automated hardware design from higher-level abstraction is becoming more and more important. However, for many high-level programming languages, such as C/C++, the description of bitwise access and computation is not as direct as hardware description languages, and hardware synthesis of algorithmic descriptions may generate sub-optimal implement-tations for bitwise computation-intensive applications. In this paper we introduce a bit-level transformation and optimization approach to assisting hardware synthesis of algorithmic descriptions. We introduce a bit-flow graph to capture bit-value information. Analysis and optimizing transformations can be performed on this representation, and the optimized results are transformed back to the standard data-flow graphs extended with a few instructions representing bitwise access. This allows high-level synthesis tools to automatically generate circuits with higher quality. Experiments show that our algorithm can reduce slice usage by 29.8% on average for a set of real-life benchmarks on Xilinx FPGAs. In the meantime, the clock period is reduced by 13.6% on average, with an 11.4% latency reduction.

1. INTRODUCTION

Bitwise operations are used extensively in many application domains, such as cryptography and telecommunications, etc. However, for applications written in high-level programming languages and executed on general-purpose processors, accessing and computing bit-values are relatively expensive, and bit-level parallelism is not well exploited. This is mainly due to the lack of support in target machines, as well as high-level programming languages, such as C/C++. Most general-purpose processor architectures and high-level programming languages do not support bitwise memory access and require a series of load/shift/mask/store instructions to implement simple bitwise operations, such as bit accessing and bit setting.

Customized hardware accelerators provide a promising approach to assisting general-purpose processors in exploiting performance of bitwise computation-intensive applications. Today, we can put more than one billion transistors in a single chip [1], and modern FPGAs allow users to exploit parallelism in applications by hundreds of thousands of logic cells and prefabricated IPs [2]. As RTL coding time is increasingly recognized as a significant component of the overall effort to solution, automated design processes and tools which compile higher-level abstraction into optimized hardware are gaining more and more popularity [3-6]. However, high-quality implementations are difficult to achieve automatically, especially when the description of the functionality is written in a high-level software programming language. For bitwise computation-intensive applications, one of the main difficulties is the lack of bit-accurate descriptions in high-level software programming languages. The wide use of bitwise operations in certain application domains calls for specific bit-level transformation and optimization to assist hardware synthesis of algorithmic descriptions.

Figure 1 shows a motivational example with a bit reversing function. The C description of this algorithm is shown in Figure 1(a), where the *bit_reverse* function takes a 32-bit integer as input and yields an output in the reverse bit order. The data-flow graph for the unrolled function is shown in Figure 1(b), while the optimal implementation is shown in Figure 1(c). We can clearly see that the direct implementation based on the data-flow graph would use many more logical components and also have a longer latency compared to the optimal one, which only uses 32 wires to link the bits directly in the reverse order.

```
int bit_reverse(int input)
{
  int i, output = 0;
  for (i = 0; i < 32; i++ )
    output |= (((input>>i)&1) << (31-
i));
  return output;
}
```

(a) C code for the *bit_reverse* function.
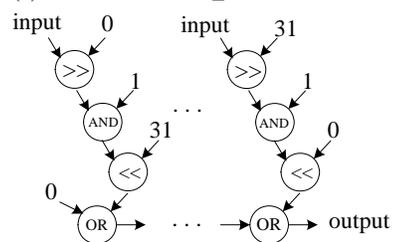
(b) Data-flow graph for unrolled *bit_reverse* function.
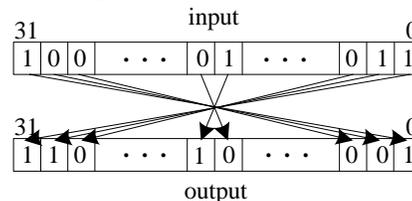
(c) Optimal implementation for the *bit_reverse* function.

Figure 1: The bit reversing function.

We can see from the example that efficient bit-level transformation and optimization for operations in algorithmic descriptions will lead to a much more direct and compact description. This will help the high-level synthesis to generate better RTL designs for bitwise computation-intensive designs, and thus achieve better final implementations. Otherwise, in the absence of such optimization, the synthesis process can be misled by inaccurate area and timing estimation and thus generate suboptimal microarchitecture. It is often too late for the downstream RTL or logic synthesis and optimization techniques to make up for the QoR loss caused by the mistakes during compiler transformations.

In this paper we propose a novel compiler optimization approach to automatically generate bitwise operations for hardware synthesis from the algorithmic descriptions in high-level programming languages. Specifically, we extend the data-flow graph with two operations (instructions) representing bitwise access to greatly facilitate the hardware synthesis to synthesize algorithmic description into efficient hardware. We propose an intermediate representation called the bit-flow graph (BFG) to analyze and optimize bitwise operations, and the optimized BFG is transformed to the extended data-flow graph for hardware synthesis. To our knowledge, this is the first work to systematically analyze and optimize bitwise operations to assist hardware synthesis of algorithmic description. Experiments show that our approach can achieve a 29.8% area reduction, 13.6% clock period reduction and 11.4% latency reduction on average for a set of real-world applications.

The remainder of the paper is organized as follows: In Section 2 we review the related work. Section 3 presents the problem statement. In Section 4 we describe our bit-level transformation and optimization approach. Section 5 presents experimental results. Section 6 concludes the discussion of current work and proposes future directions.

2. RELATED WORK

In this section we discuss previous work on optimization for bitwise computation-intensive applications.

Modern optimizing compilers can perform a series of transformation passes (typically in the form of peephole optimizations) to simplify logical operations [7, 8]. For example, algebraic simplifications and reassociation can be applied to Boolean and structure bit-field types using logical computation properties. Constant folding evaluates constant expressions at compile time and replaces variable references with constants. One of the main characteristics of these techniques is that they manipulate operands mainly at byte/word level and rarely analyze bit-value flow information. This is usually sufficient when the target is a general-purpose microprocessor. However, for application-specific hardware implementations, we may miss many important optimization opportunities that potentially lead to better solutions, especially for bitwise computation-intensive applications.

Several modern processors extend their instruction sets to accelerate bitwise operations. The counting-leading-ones, counting-leading-zeros and counting-set-bits instructions are such extensions existing on some general-purpose processors. Hilewitz et al. [9] conjectured that the most powerful primitive bit-level operation might be the bit matrix multiply (BMM) instruction, which currently is found only in supercomputers like Cray[10]. They also proposed new instructions that implement simpler BMM primitive operations. However, the current code-generation techniques for these instructions mainly seek for special patterns, and efficiently taking use of these instructions still much relies on hand-coded assemble codes.

In the logic synthesis field, much research has been conducted to simplify logical expressions [11]. However, when using high-level programming languages, the bit-value accessing, computing and storing are indirectly represented and often require a series of load/shift/ mask/store instructions. If the bitwise computation is not well analyzed and optimized during the high-level synthesis step, the resulting RTLs can be suboptimal. This would impose difficulty for the downstream RTL/logic synthesis and optimization to make up the QoR degradation.

Some hardware modeling languages extend high-level software programming languages, and most of them support bit-accurate description. For example, SystemC [12], which is a popular modeling language based on C++, introduces bit-accurate data types to support description for bit-level access and computation. Some related works also extend a base sequential language with direct bit- manipulation for both software and hardware. For example, [13] introduces a new object-oriented language called Lime, which can be compiled for JVM or into a synthesizable hardware description language. It provides explicit bit-numeration to describe bitwise operations. Nevertheless, most software algorithms and a large amount of legacy code are still written in high-level software programming language.

In contrast to the previous work, our approach aims at providing bit-level transformation and optimization to assist hardware synthesis of algorithmic descriptions. Since hardware directly supports bit-value accessing and storing, while a large amount of software legacies still use load/ shift/mask/store instructions to represent bitwise operation, there is a gap between function description in high-level programming languages and hardware synthesis. To deal with this problem, we propose a new intermediate representation for bitwise operations. It will facilitate bit-value analysis and provide a platform to take advantage of the existing logical expression simplification techniques before hardware synthesis.

3. PROBLEM STATEMENT

In this section we formalize the problem of transforming the data-flow graph (DFG) to greatly improve the area and performance of the generated circuits for bitwise computation-intensive applications.

We define the BO-DFG as a data-flow graph which contains only the basic logical, shift and conversion operations, as listed in Table 1. These operations will be referred to as bitwise operations in this paper. These operations are the ones usually supported by high-level programming languages and compiler intermediate

representations. Given a DFG derived from a software description, we extract the BO-DFGs in it for further analysis and optimization.

In order to represent direct bitwise accesses, we further introduce three instructions (or operations) into DFG, as shown in Table 2.

Table 1: Basic logical, shift and conversion operations.

Class	Operations
Logical	AND, OR, XOR, NOT
Shift	Shift left, Logical / Arithmetic shift right
Conversion	Truncate, Zero-extension, Sign-extension

Table 2: Additional instructions to represent bitwise access.

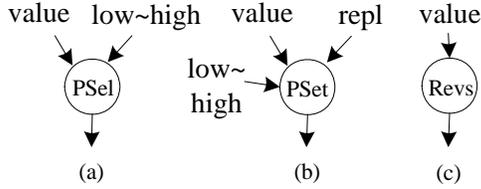| Instruction	Symbol	Operands and Outputs
part_select	PSel	output = part_select(value, low, high)
part_set	PSet	output = part_set(value, repl, low, high)
Reverse	Revs	output = reverse (value)

Figure 2: Graphic representation of *part_select*, *part_set* and *reverse* operations.

Their semantics are defined as follows:

1) The *part_select* instruction selects the *low* through *high* bits from *value* as the output. The operand *high* should be equal to or greater than the operand *low*. We use *PSel* to symbol it. The graph representation is shown in Figure 2 (a).

2) The *part_set* instruction replaces the bits between *low* and *high* (inclusive) of *value* with the lowest (*high* – *low* + 1) bits from *repl*, and output the result. That is the 0th bit in *repl* replaces the *low* bit in *value* and etc. up to the *high* bit. The operand *high* should be equal to or greater than the operand *low*, too. We use *PSet* as its symbol, and the graph representation is shown in Figure 2 (b).

3) The *reverse* instruction outputs all the bits from *value* in the reverse order. We use *Revs* as its symbol, and the graph representation is shown in Figure 2 (c).

The data-flow graph for the *bit_reverse* function with the *reverse* instruction is shown in Figure 3, which is much more compact than the one shown in Figure 1 (b).
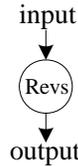

Figure 3: The data-flow graph for the *bit_reverse* function with the *reverse* instruction.

Let *G* (*V*, *E*) be a BO-DFG which only consists of the operations listed in Table 1. We define two types of cost functions associated with *G*: delay cost (D-cost) and component cost (C-cost). D-cost is determined by the longest path delay of *G*. C-cost, on the other hand, is the weighted sum of all nodes and edges. The nodes

and edges form a component set (ComSet), and each component has an associated weight that corresponds to the estimated area of the component.

$$C - Cost(G) = \sum_{com_i \in ComSet} Count(G)_{com_i} * Weight_{com_i}$$

For convenience, we define two graphs *G* and *G'* as semantically equivalent if all outputs of the two circuits implementing graph *G* and graph *G'* are identical under any combination of input values. Then the problem can be formalized as follows:

Problem: Given a BO-DFG *G*(*V*, *E*) derived from a high-level description, which only contains the basic bitwise operations in Table 1, transform *G* into a semantically equivalent graph *G'* extended with *part_select, part_set* and *reverse* instructions listed in Table 2 so that C-cost or D-cost is minimized.

We believe that solving the above problem efficiently will greatly benefit the high-level synthesis for bitwise computation-intensive designs. Otherwise, in the absence of such transformation, the area and timing estimation may be inaccurate, and the high-level synthesis process can be misled and thus generate suboptimal microarchitecture. The downstream RTL or logic synthesis and optimization techniques are often too late to make up for the QoR loss caused by the mistakes during the early stage.

In the subsequent sections, we propose an approach that derives the bitwise access and operation information from the shift and mask (and/or) operations in software description and simplifies the bitwise operations in this intermediate representation. Then a data-flow graph with explicit bitwise accesses is generated. With our approach, the cost can always be reduced for hardware implementations.

4. BIT-LEVEL TRANSFORMATION AND OPTIMIZATION

In this section we present our approach of analyzing and optimizing bitwise operations. The algorithm outline is shown in Figure 4. Given a data-flow graph, our algorithm will first extract the BO-DFGs in it and construct bit-flow graphs for them, which will be described in detail in Subsection 4.1. Then a series of transformations is performed to reduce some obvious computation redundancy, as presented in Subsection 4.2, and if needed, various logical expression simplification techniques can also be taken. Finally, each BFG is transformed back to a data-flow graph, as presented in Subsection 4.3. The algorithm complexity will be analyzed in Subsection 4.4.
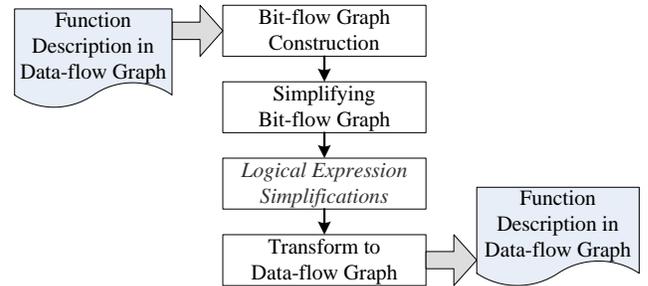

Figure 4: Algorithm flow.

4.1 BFG Construction

We propose a new intermediate representation here for BO-DFGs. We observe that for bitwise logical operations, the computation for each bit is independent from other bits in the same variable. Thus we can transform the data-flow graph, viewing each bit as independent element and simplifying the representation. We propose an intermediate representation called the bit-flow graph (BFG) to keep track of various types of bit-value information, such as whether the bit is a constant, whether it is equivalent to a bit from another variable, and the operation to compute this bit, etc.

BFG is similar to the data-flow graph, except that each node or edge in a BFG represents the data dependency for the width of only one bit; that is, BFG is a directed graph which shows the bit-value dependencies between operations. The main data structure of a BFG node is shown in Figure 5.
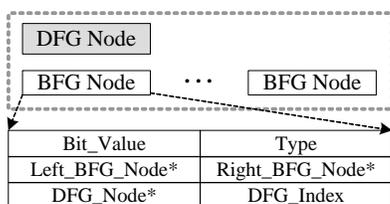

Figure 5: BFG node description.

The nodes in BFG contain the following types: AND, OR, XOR, NOT, SET, VARIABLE and CONSTANT. Nodes of types VARIABLE and CONSTANT are leaf nodes, which are input nodes of the whole graph, and each of them has one output port. Nodes of types AND, OR and XOR represent one-bit and/or/xor operations respectively, each with two input ports and one output port. NOT nodes represent one-bit not operation, with one input port and one output port. SET node has one input port and one output port, representing that the bit-value from input port flows to the output port. The edges connect output ports and input ports, representing the flow of bit-values. The BFG data structure also contains two fields indicating the bit's position in the original DFG: *DFG Node Pointer* and *DFG Index*. If the *DFG Node Pointer* is nonzero, it represents that the BFG node corresponds to the (*DFG Index*)th bit in the DFG node pointed by the *DFG Node Pointer*.

When building BFG for a BO-DFG, we traverse the BO-DFG in postorder, i.e., visiting all operands of an operation before visiting the operation node itself. For each BO-DFG node, we build BFG nodes and corresponding edges in the following manner:

1) If the current BO-DFG node is a leaf node, we build a BFG node for each bit. For a BO-DFG node with width N, N BFG nodes will be created and key information will be recorded in their data structures, such as the bit's value, the corresponding BO-DFG node and the bit order number.

2) If the current node is not a leaf node, we will check its operation type and construct corresponding BFG nodes. We take an N-bit SHL (left shift) operation as an example. Assume that the shift amount is constant M. First, N BFG nodes are created. Since the lowest M bits of the result will be zero, the first M BFG nodes are all set to be zero. Then for the left ($N - M$) BFG nodes, the operation type for them is set to be SET and the input edges are connected to the corresponding BFG nodes of the shift variable. Figure 6 shows the result.
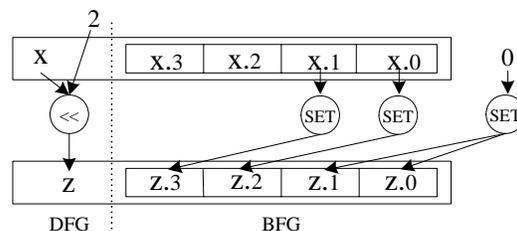

Figure 6: A 4-bit shift operation example: z =SHL(x, 2).

The examples of BFG node construction for each class of operations are shown in Figure 7. BFG node construction for OR/XOR/NOT is similar to the construction for AND in Figure 7 (a); BFG node construction for logical right shift and arithmetic right shift are similar to the one for left shift in Figure 7 (b); BFG nodes construction for truncation and signed extension is similar to the one for zero-extension in Figure 7 (c). Figure 8 (a) shows the generated BFG for the *bit_reverse* function after direct construction.

4.2 Simplifying BFG

After a BFG is built, we simplify the BFG to eliminate redundant computations by traversing the BFG in postorder and applying the following transformation rules:

1) For a SET/NOT node, if its input node (i-node) is a SET node, we change its input node to i-node's input node (see Figure 9 (a));

2) For an AND/OR/XOR node, if one of its input nodes is a SET node (s-node), we change the input node to s-node's input node; (see Figure 9 (b));

3) For a SET/NOT node, if its input node is a CONSTANT node, we replace the SET node with the corresponding CONSTANT node (see Figure 9 (c));

4) For an AND/OR/XOR node, if both of its input nodes are CONSTANT nodes, we replace the node with a CONSTANT node, whose value is the calculated result (see Figure 9 (d));

5) For an AND node, if one of its input nodes is CONSTANT ZERO, we replace the AND node with CONSTANT ZERO (see Figure 9 (e)); If it is a ONE, we replace the AND node with a SET node of the other input node;

6) For an OR node, if one of its input nodes is CONSTANT ONE, we replace the OR node with CONSTANT ONE; If it is a ZERO, we replace the OR node with a SET node of the other input node

7) For an XOR node, if one of its input nodes is a CONSTANT ZERO, we replace the XOR node with the other input node. If it is a ONE, we replace the XOR node with the opposite of the other input node;

8) For a NOT node, if its input node is a CONSTANT node, we replace the NOT node with the opposite CONSTANT node of its input node.
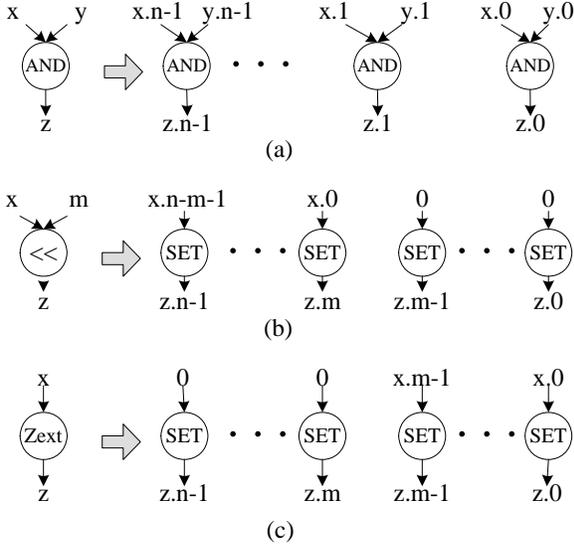
Figure 7: BFG node construction. "x.i" represents the ith bit of x. (a) shows the BFG node construction for z = x & y (n bits); (b) shows the BFG node construction for z = x << m (x is n-bit wide); (c) shows the BFG node construction for z = zero_ext x (z is n-bit wide; x is m-bit wide).
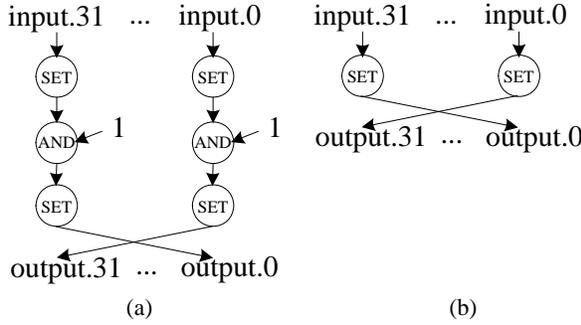

Figure 8: The *bit_reverse* function in BFG. (a)The BFG after direct construction; (b)The BFG after simplification.
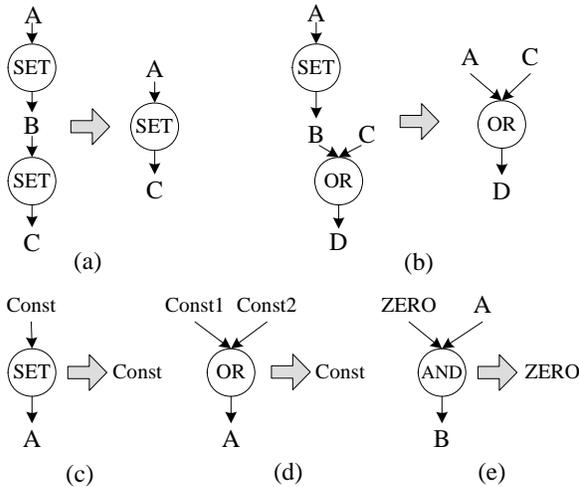

Figure 9: Examples of transformation rules.

Among these rules, Rules 1 to 3 are like bitwise copy-propagations, while the others belong to bitwise

peephole optimization using the computation rules. We can easily prove that the simplifying process of iteratively applying the upper rules is the process of reducing BFG C-cost. The D-cost will not be increased in this process, and sometimes it can be reduced as well. The simplified BFG representation of the *bit_reverse* function is shown in Figure 8 (b), which describes direct bit settings.

After building up the BFG, logical expression simplification techniques can be performed on this graph. Existing approaches can be used to optimize the BFGs for different purposes. Since making use of the existing optimizations is not the main point of this paper, we will not further elaborate this step.

4.3 Transforming BFG to Extended DFG

In this subsection, we introduce the process of transforming a function in BFG form back to DFG form extended with the *part_select*, *part_set* and *reverse* operations.

For convenience of explanation, we define *consecutive BFG nodes* as follows.

Consecutive BFG nodes are the BFG nodes:

i) Representing consecutive bits in a variable or constants in the original DFG, either in the forward order or in the reverse order;

Or ii) which have the same operations and their input nodes are consecutive BFG nodes separately.

See Figure 10 for example. Since bits 0~1 of x are consecutive, two SET nodes (in the dotted border) representing setting bit 1 of z with bit 0 of x and setting bit 2 of z with bit 1 of x are consecutive BFG nodes, and we can generate one single operation: t = *part_set (0, x, 1, 2)* for them. Bit 0 of y is not consecutive with bit 1 of x, so we should generate a separate operation: z = *part_set* (t, y, 3, 3) to represent setting bit 3 of z with bit 0 of y.
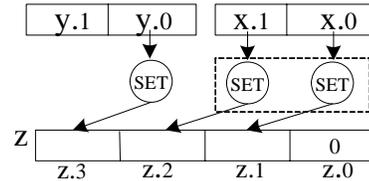
Figure 10: Example of consecutive BFG nodes.

We call the *Trans_to_DFG* function for each DFG root to transform the corresponding BFG back to an extended DFG and then replace the original DFG. The algorithm of *Trans_to_DFG* is shown in Figure 11. In the algorithm, we firstly traverse the input DAG and mark the vectors of consecutive BFG nodes corresponding to the DFG root. Then we call *Part_trans_to_DFG* function iteratively to generate DFG for each vector of consecutive BFG nodes. In *Trans_to_DFG* , the *size_of* function returns the number of nodes in a node vector. The *gen_node* function generates and returns a DFG node. The first argument of *gen_node* indicates operation type of the node and the following arguments are the operands of the operation.

The *Part_trans_to_DFG* algorithm is also shown in Figure 11. It takes a vector of consecutive BFG nodes as input, generates the corresponding DFG for the vector of BFG nodes, and returns the root of the

generated DFG. It generates DFG nodes according to the types of the input BFG nodes: 1) If the input BFG nodes are CONSTANT nodes, it generates a constant DFG node with the value of these bits; 2) If the input BFG nodes represent consecutive bits from a variable, it firstly checks whether they represent all the bits form the variable. If not, a *part_select* node is generated. Then it checks whether the bits from the variable are in reverse order to decide whether to generate a *reverse* node; 3) If the input BFG nodes are SET nodes, the *Part_trans_to_DFG* function is called with a new vector containing the inputs of these BFG nodes; 4) If the input BFG nodes are NOT nodes, the *Part_trans_to_DFG* function is called to generate a DFG for the operands. Then it generates and returns a NOT DFG node; 5) If the input BFG nodes are AND/OR/XOR nodes, the DFGs for the left and right operands are generated by the *Part_trans_to_DFG* function and a corresponding AND/ OR/XOR DFG node is generated and returned.

This algorithm can be followed by a local common-subexpression elimination pass to reduce potential redundancy.

Algorithm 1: Trans_to_DFG

Input: *rootVector — a vector of BFG nodes for a DFG root*
Output: a pointer to the root of the generated DFG
Trans_to_DFG
begin
   pRoot ← a pointer to a DFG node representing ZERO;
   i ← 0;
   while *i* < sizeof(*rootVector*) **do**
      consNodeVector ← consecutive BFG nodes starting from the ith node in rootVector;
      tmpNode ← Part_trans_to_DFG*(consNodeVector);*
      len ← size_of*(consNodeVector);*
      //Generate part_set to set the i ~ i+len-1 bits of newRoot.
      pRoot ← gen_node*(PART_SET, pRoot,   tmpNode,   i,   i+len-1);*
      i ← i + length;
   endwhile
   return *pRoot*;
end

Algorithm 2: Part_trans_to_DFG

Input: *nv — a vector of consecutive BFG nodes*
Output: a pointer to the root of the generated DFG
Part_trans_to_DFG
begin
   len ← size_of(*nv*);
   switch (*nv*[0]-> type)
   case CONSTANT **then**
      return gen_node (CONSTANT, value_of(*nv*))*;*
   case VARIABLE **then**
      if *nv* contains BFG nodes corresponding to all the bits of a variable **then**
         tmpNode ← nv[0]->DFG_Node;
      elif *nv* contains BFG nodes representing consecutive bits from a variable in reverse order **then**
         tmpNode ← gen_node(PART_SELECT,
               nv[0]->DFG_Node, *nv*[*len*-1]->DFG_Index,
               nv[0]->DFG_Index);
      else
         tmpNode ← gen_node(PART_SELECT,

nv[0]->DFG_Node, *nv*[0]->DFG_Index,
nv[*len*-1]->DFG_Index);
      endif
      if *nv* contains BFG nodes representing consecutive bits from a variable in reverse order **then**
         tmpNode ← gen_node (REVERSE, *tmpNode*);
      endif
      return *tmpNode;*
   case SET **then**
      nv_set ← All the operands of nodes in *nv;*
      return Part_trans_to_DFG (*nv_set*);
   case NOT **then**
      op_nv ← All the operands of nodes in *nv*;
      op ← Part_trans_to_DFG(*op_nv*);
      return gen_node (NOT, *op*);
   case AND/OR/XOR **then**
      //Generate DFG for the left and right
      //operands recursively.
      nv_left ← All the left operands of nodes in *nv*;
      nv_right ← All the right operands of nodes in *nv*;
      l_op ← Part_trans_to_DFG(*nv_left*);
      r_op ← Part_trans_to_DFG(*nv_right*);
      //Generate DFG node for the computation.
      //*nv*[0]->type indicates the operation type.
      return gen_node (*nv*[0]->type, *l_op*, *r_op*);
   endswitch
end

Figure 11: Pseudo code of *Trans_to_DFG* and *Part_trans_to_DFG* Algorithms.

With the above algorithms, the resulted data-flow graph for the motivational example of the *bit_reverse* function is the same as in Figure 3, which only uses one *reverse* operation.

4.4 Algorithm Complexity

Let *N* denote the number of nodes in the given data-flow graph. Let *W* be the width of each node (If the width of the variables are different, we can take the largest one). According to the BFG construction method described in Subsection 4.1, there are at most (*W* * *N*) BFG nodes, and the computation complexity in step 1 is O (*W* * *N*). The computation complexities of simplifying BFG and transforming BFG back to DFG are also O (*W* * *N*). The overall complexity is O (*W* * *N*).

5. EXPERIMENTAL RESULTS

In this section, we describe our experimental environment and results of a set of real-life designs. We have implemented our algorithm in the LLVM compiler infrastructure [14]. Two families of intrinsic functions — llvm.part.select and llvm.part.set, are available in the LLVM virtual instruction set [15]. Currently, LLVM does not provide any specific analysis or optimization on these intrinsics. Nevertheless, they are very useful for expressing how bit values flow from one variable to another, and our *part_select*, *part_set* and *reverse* instructions can be directly mapped to these intrinsics.

For hardware implementation, we use a leading-edge commercial C-to-gates synthesis tool [6] to synthesize our optimized design into RTLs. The tool can take behavior-level and system-level SystemC/C/C++ as input descriptions and is able to target either ASIC or FPGA platforms. It also provides an option to synthesize the LLVM byte code. In this experiment we

use Xilinx ISE 9.2 toolset to synthesize the RTL outputs and perform placement and routing onto Xilinx Virtex-4 FPGAs [2].

Our bit-level optimization approach will benefit from proper compiler transformations, some of which are usually not default options when aiming at general-purpose processors. We apply constant propagation for constant array elements and loop unrolling when they could be beneficial to the bit-level optimization.

We also perform several local transformations in advance to transform certain frequent patterns to *part_select* or *part_set* for further speeding up the bitwise analysis process (see Table 3).

Table 3: Local transforming patterns.

No.	Patterns
1	t1 = Logical_shift_right (x, a) ➔ t1 = part_select (x, a, Width(x) – a)
2	t1 = Logical_shift_right (x, a); t2 = And (t1, 2^ b -1) ➔ t2 = part_select(x, a, a + b - 1)
3	t1 = And (x , 2^a -1); t2 = Shift_left (t1, b) ➔ t2 = part_set (0, x, b, b + a - 1)
4	t1 = And (x, 2^a – 1); t2 = Shift_left (t1, b); t3 = And (y, 2^(b+a-1) – 2^b); t4 = Or (t2, t3) ➔ t4 = part_set (y, x, b, b+a-1)

Table 4: Experiment results for *bit_reverse* function.

Function	Slices		CP (ns)		Latency (cycle)	
	ORIG	BTO	ORIG	BTO	ORIG	BTO
bit_reverse	43	0	3.45	0.00	3	0

Firstly, we have a look at the effect of our algorithm on the motivating example of *bit_reverse* function. According to Table 4, we can easily generate the intended optimal implementation for the *bit_reverse* function. The resulting RTL only uses wires to connect the bits in the reverse order using zero slices. Here, the "BTO" columns refer to the experiment results using our bit-level optimization, while the "ORIG" columns refer to the ones generated without our optimization. CP refers to clock period.

We further take a set of real-life designs for experiment from the cryptography and telecommunication domains, as listed in Table 5. The program profiles including the number of C lines, the number of LLVM operations, and the percentages of the bitwise operations are also shown in Table 5. Here the bitwise operations refer to the operations listed earlier in Table 1, and the percentages are collected from LLVM intermediate representations of these benchmarks. We can see that the five benchmarks are all bitwise computation-intensive applications, with at least 46.1% bitwise operations.Table 6 presents the bit-level transformation effects on the set of designs. The "BTO"columns are collected from LLVM intermediate representations of these benchmarks optimized with our bit-level transformation pass, while the "ORIG" columns are from the ones optimized without our pass. We notice that our bit-level transformation algorithm will often generate narrower logical instructions. For example, when analyzing "(x >> 8) & y" by our pass, the bit-level transformation algorithm will see the highest 8 bits of x are all zeros. Supposing x and y are both 32-bit wide, the algorithm can generate a 24-bit AND operation instead of a 32-bit one. In order to see the exact reduction of logical operation units, we take the bit width of logical operations in consideration and collect how many bit logical operations are used for each benchmark, as shown in the "Unit Logical OP" columns, which refer to the number of unit And/Or/Xor/Not operations. The "Shift" columns refer to the shift operations as described in Table 1, and the "Bit Access" columns refer to the direct bit-accessing operations introduced in Table 2. We should notice that in many cases, we need one *part_select* and one *part_set* instructions to describe a bit connection. Although the numbers of bit- accessing operations of these benchmarks seem to berelatively large, many of these operations will be turned into wires and significant amounts of parallelism exists in these operations.

Table 5: Benchmark descriptions.

Benchmark	C Lines	LLVM OP	Bitwise OP Percentage	Description
GSM_pack	123	441	72.8%	The post-stage of GSM 06.10 encoder (pack wave).
GSM_unpack	125	331	81.6%	The pre-stage of GSM 06.10 decoder (unpack wave).
MD5	504	1468	49.9%	A widely used cryptographic hash function.
AES	1318	3342	46.1%	A popular algorithm used in symmetric key cryptography.
3DES	390	3975	89.3%	A popular algorithms used in symmetric key cryptography.

Table 6: Bit-level optimization effects on the LLVM intermediate representations of the benchmarks.

Benchmark	Unit Logical OP			Shift		Bit Access	Other Operations		
	ORIG	BTO	Reduction	ORIG	BTO	BTO	ORIG	BTO	Reduction
GSM_pack	12136	64	99.5%	124	0	270	123	120	2.4%
GSM_unpack	11328	0	100.0%	90	0	193	64	61	4.7%
MD5	11584	6592	43.1%	234	0	283	872	736	15.6%
AES	31908	14916	53.3%	441	0	3858	1897	1801	5.1%
3DES	29584	588	98.0%	880	0	1164	1090	422	61.3%
Average	19308	4432	78.8%	354	0	1154	809	628	17.8%

We can see from the result that our bit-level optimization algorithm can significantly reduce the unit logical operations used in the applications. For GSM_unpack, our algorithm can reduce all of the logical operations. For GSM_pack and 3DES designs, our algorithm can reduce 99.5% and 98.0% unit logical operations separately. For AES and Md5 designs, our bit-level optimization algorithm can also reduce 53.3% and 43.1% unit logical operations. On average, 78.8% unit logical operations and all the shift operations in these benchmarks can be reduced by our bit-level optimization algorithm. Other operations of these benchmarks are reduced by 17.8% on average, and most of the reduction is from the conversion operations listed in Table 1. We can see from the comparisons that our algorithm can efficiently transform unnecessary logical and shift instructions into direct bit-accessing instructions, without increasing other operations.

Finally, we compare the generated implementations optimized by our pass with the ones generated without our pass. Table 7 presents the results of the experiments, where the "BTO" columns refer to the experiment results using our bit-level optimization. The "ORIG" columns refer to the results which are generated by normal flow without the bit-level optimization and optimized for speed under area constraints. On average, our pass can lead to a 29.8% improvement of area, 13.6% clock period reduction and 11.4% latency optimization for these benchmarks. The experiments show that for bitwise computation-intensive applications, our algorithm can help the hardware synthesis to generate better implementations automatically.

Table 7: Experimental results.

Benchmark	Slices		CP (ns)		Latency (cycle)	
	ORIG	BTO	ORIG	BTO	ORIG	BTO
GSM_pack	1301	624	4.78	3.85	68	56
GSM_unpack	655	304	4.47	3.14	15	14
MD5	2714	2348	13.52	12.92	195	194
AES	2799	2383	4.38	3.77	122	88
3DES	885	754	5.99	5.99	70	67
Average Reduction		29.8%		13.6%		11.4%

6. CONCLUSION

In this paper we have introduced a bit-level transformation and optimization approach to assist hardware synthesis of arithmetic descriptions for functions with a large amount of bitwise operations. The proposed bit-flow graph can record and propagate bit-value information, allowing bitwise analysis and optimizations. The optimized BFG is transformed back to DFG extended with a few instructions representing bit accessing clearly, so that hardware synthesis of algorithmic description can generate corresponding hardware directly. Experiments show that our algorithm can reduce slices by 29.8%, reduce clock period by 13.6% and reduce latency by 11.4% on average for the benchmarks. In the future, we will consider exploring other application domains and supporting transformation for shift operations with variable shift-amount arguments to gain further improvement.

8. REFERENCES

[1] "International Technology Roadmap for Semiconductors, 2007 Edition", Semiconductor Industry Association.

[2] http://www.xilinx.com, Xilinx Website.

[3] T. Bollaert, "Catapult Synthesis: A Practical Introduction to Iterative C Synthesis", in *High-Level Synthesis: From Algorithm to Digital Circuit*, Eds. P. Coussy and A. Morawiec: Springer Publishers, 2008.

[4] M. Meredith, "High-level SystemC Synthesis with Forte's Cynthesizer", in *High-Level Synthesis: From Algorithm to Digital Circuit*, Eds. P. Coussy and A. Morawiec: Springer Publishers, 2008.

[5] S. Aditya and V. Kathail, "Algorithmic Synthesis Using PICO", in *High-Level Synthesis: From Algorithm to Digital Circuit*, Eds. P. Coussy and A. Morawiec: Springer Publishers, 2008.

[6] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "AutoPilot: A Platform-Based ESL Synthesis System", in *High-Level Synthesis: From Algorithm to Digital Circuit*, Eds. P. Coussy and A. Morawiec: Springer Publishers, 2008.

[7] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann Publishers, 1998.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA Addison-Wesley Longman Publishing Co., Inc, 2006.

[9] Y. Hilewitz, C. Lauradoux, and R. B. Lee, "Bit Matrix Multiplication in Commodity Processors", in *19th IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2008.

[10] "Cray Assembly Language (CAL) for Cray X1 Systems Reference Manual, version 1.2", Cray Inc. 2003.

[11] G. De Micheli, *Synthesis and Optimization of Digital Circuits*: McGraw-Hill Higher Education, 1994.

[12] Open SystemC Initiative, "IEEE Standard SystemC® Language Reference Manual", 2006.

[13] S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, "Liquid Metal: Object-Oriented Programming across the Hardware/Software Boundary", in *Proceedings of the 22nd European conference on Object-Oriented Programming*, Paphos, Cypress, 2008.

[14] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", in *Proceedings of the international symposium on Code Generation and Optimization*, Palo Alto, California, 2004.

[15] "LLVM Language Reference Manual", http://www.llvm.org/docs/LangRef.html.